Source of Acquisition
NASA Ames Research Center

Inspection and Verification of Domain Models with PlanWorks and Aver

Tania Bedrax-Weiss[*†] and **Jeremy Frank**[‡] and **Michael Iatauro**[§] and **Conor McGann**[¶]
Computational Sciences Division
NASA Ames Research Center, MS 269-2
miatauro@email.arc.nasa.gov
Moffet Field, CA 94035

Introduction and Motivation

When developing a domain model, it seems natural to bring the traditional informal tools of inspection and verification, debuggers and automated test suites, to bear upon the problems that will inevitably arise. Debuggers that allow inspection of registers and memory and stepwise execution have been a staple of software development of all sorts from the very beginning. Automated testing has repeatedly proven its considerable worth, to the extent that an entire design philosophy (Test Driven Development) has been developed around the writing of tests.

Unfortunately, while not entirely without their uses, the limitations of these tools and the nature of the complexity of models and the underlying planning systems make the diagnosis of certain classes of problems and the verification of their solutions difficult or impossible.

Debuggers provide a good local view of executing code, allowing a fine-grained look at algorithms and data. This view is, however, usually only at the level of the current scope in the implementation language, and the data-inspection capabilities of most debuggers usually consist of on-line print statements. More modern graphical debuggers offer a sort of tree view of data structures, but even this is too low-level and is often inappropriate for the kinds of structures created by planning systems. For instance, goal or constraint networks are at best awkward when visualized as trees. Any any non-structural link between data structures, as through a lookup table, isn't captured at all. Further, while debuggers have powerful breakpointing facilities that are suitable for finding specific algorithmic errors, they have little use in the diagnosis of modeling errors.

Automated testing can take several forms, few of them convenient. Writing tests explicitly in code can require deep knowledge of the system in which the model is going to be executed, are therefore not portable to other planning systems, even closely related ones, and will break with changes in the underlying system or the model, adding to the required maintenance work. Tests written at this level will also have to be much more verbose than those written at a higher level of abstraction.

EUROPA(Frank & Jónsson 2003), the predecessor to EUROPA$_2$, as part of its test suite, captured the output of the final plan and compared it against a known-good output. This proved to be quite brittle, since changes to the planner, plan database, model, or heuristics could dramatically alter the output without implying a bug, and hand-verifying output for the new known-good was both tedious and labor intensive. The known-good method also suffers from a limitation of scope—it looks only at the output, and in the case of planning and model rule execution, the path to the final plan is at least as important.

Another verification technique that EUROPA employed was an examination of the final constraint network to ensure compliance with the rules of the model. While suffering from the output-scope problem, it also only detects errors in the code that executes model rules which, while significant, is only one of a plurality of components. This technique also only checks the constraint network's compliance with the model, not the executed model's compliance with the intended model.

Clearly, there is a gap between what traditional tools can provide and what is necessary to debug and test planning systems efficiently. To this end, we have built two tools: *PlanWorks*, a visualization and query tool for plan inspection and *Aver*, a language for the specification of automated tests.

This paper is organized as follows. We first describe some fundamentals of the EUROPA$_2$ constraint-based planning system. We then describe our debugging tool, PlanWorks. We cover in light detail its views and query tools. We then describe our test specification language, Aver. We describe its method of asserting properties of plans with queries and boolean comparisons. We then describe the use of these tools to verify the description of a sample problem domain and instance, the pipesworld, in which we cover test composition, a test failure, its investigation with PlanWorks, and confirmation of the fix with both PlanWorks and the automated test. Finally, we discuss future work for both tools.

The EUROPA$_2$ Paradigm

The context in which these tools have been developed is EUROPA$_2$, which provides plan database services that en-

*Authors listed in alphabetical order.
†QSS Group, Inc.
‡NASA
§QSS Group, Inc.
¶QSS Group, Inc.

able the integration of automated planning into a wide variety of applications.

A detailed discussion of the EUROPA$_2$ paradigm is beyond the scope of this paper, but a brief discussion is included here. A *plan* is a complete enumeration of the states necessary to achieve a set of goal states from a set of initial states which satisfies the constraints of a planning domain and problem instance. In EUROPA$_2$, states are represented as *predicates*, each of which has a name, start time, end time, duration, and zero or more parameters. Each instance of a predicate in a plan is represented by a *token* and the parameters, timepoints, and duration of the predicate are represented by *variables*. Predicates are associated with *classes* that represent types of *objects*, with specializations like *timelines*, which require that their sequences of states be totally ordered, or *resources*, which allow concurrent states, but require that rules about consumption and production rates and resource levels be obeyed. During planning each token is assigned to an object. *Domain rules* are assertions that if a predicate P is in the plan, then other predicates Q_i must also be in a plan and are related to P by *constraints* among the variables of the predicates. Domain rules may also assert that resources are impacted by predicates; resource impacts are called *transactions* and also have variables that represent them.

It is important to emphasize that EUROPA$_2$ does not implement any planning algorithm; rather, it provides services that support different planning algorithms according to the application, like maintaining plan state and evaluating plan consistency. The EUROPA$_2$ plan database maintains the current plan state and an external planner performs the search by resolving flaws through variable restrictions, which amount to operations on the plan database. As such, it can be used to support progression planners, regression planners, sequential or causal link planners, and so on. To enable this generality, EUROPA$_2$ distinguishes between *free* tokens (consequences of rules that haven't been inserted into plans), *active* tokens, and *merged* tokens. Planners can insert free tokens into plans, making them active, or co-designate free tokens with active tokens, making them merged.

PlanWorks

Introduction

PlanWorks is a browse-based system for debugging constraint-based planning and scheduling systems. It assumes a strong transaction model of the entire planning process, including adding and removing parts of the constraint network, variable assignment, and constraint propagation. A planner logs transactions and plan states for importation into a relational database that is tailored to support queries for a variety of components. *Visualization* components consist of specialized views to display different forms of data (e.g. constraints, activities, resources, and causal links). Each view allows user customization in order to display only the most relevant information. Inter-view navigation features allow users to rapidly exchange views to examine the trace of the process from different perspectives. *Transaction query* mechanisms allow users access to the logged transactions to visualize activities across the entire planning process.

PlanWorks is implemented in Java and employs a MySQL relational database back-end. It can be used either online while planning is performed or offline after capturing the entire planning process. Furthermore, PlanWorks is an open system allowing for extensions to the transaction model to capture new planner algorithms, different classes of entity, or novel heuristics. While PlanWorks was specifically developed for EUROPA$_2$, the underlying principles behind PlanWorks make it useful for many constraint-based planning systems.

Views

The first view the user is presented with is an overview of the entire planning sequence, an inverted histogram of the counts of the tokens, variables, and constraints in the plan at each step. Moving the mouse over a histogram element will reveal the the number of elements of a particular type at that step. At a glance, the user sees how the plan's size evolved throughout planning and can see patters (such as thrashing in a chronological backtracking algorithm, or local optimum in a local search planner). An indicator above each histogram bar indicates whether the data for that step is in the file system or in the PlanWorks database.

The *Timeline View* is designed to show the sequence of predicates on a timeline. Since tokens can be co-designated, the Timeline View shows the number of co-designated tokens that each token supports.

Because the EUROPA$_2$ structure can be treated as a directed graph (Objects$\rightarrow$Tokens$\rightarrow$Variables$\leftrightarrow$Constraints), it is useful to visualize the entire graph or certain subgraphs. Of particular interest are the causal tree, or *token network*, and the *constraint network*. All PlanWorks graph views use an incremental expansion method for navigation. Clicking on a node will expand all of its arcs and place in the view any connected nodes not already visible. Clicking on such an "open" node closes it, and will cause any entities to which it is related that are not connected to other open entities to be removed from the view. To assist navigation, the graph views provide "find by key" and "find path" to locate a particular entity in the graph and find a path between two entities, respectively.

The Token Network View visualizes the causal chain resulting from planner decisions and model rules. Initially only the root tokens—those created in the initial state—are visible. Expanding a token node causes the appearance of *rule* nodes, which represent the model rules that executed because of the presence of the parent token in the plan. Rule nodes can be expanded to see the text of the rule as written in the model as well as to see the tokens created through application of the rule.

The Constraint Network View begins with model invariants, objects, tokens, and instances of rule execution. Each of these entities is associated with a set of variables, which in turn are in the scope of constraints. "Opening" a starting entity will reveal its variables, each of which will reveal its constraints when opened.

The Navigator View is the union of the Token Network and Constraint Network views as well as information not

explicit in any other view. Beginning from an entity present in some other view and every immediate neighbor entity, the Navigator view allows incremental exploration of every entity connection present in the plan.

The amount of information in a plan quickly exceeds that which can be easily treated by these views, so PlanWorks offers a *Content Filter* to restrict the visual elements to those related to particular predicates, the predicates of particular objects, or predicates within a specified window of time.

Transactions

EUROPA$_2$ has a rich transaction set describing the various transformations within the plan database, constraint network, and rules engine that it uses for internal notification, but which also has value in debugging. PlanWorks offers a mechanism for querying the transactions on individual entites, of a particular type, that represent the state transformation from one step to the next, or a combination of these.

Planner Control

Planning can be quite expensive in terms of time and logging data after every planner decision only slows the process down, which can be counterproductive when one is attempting to determine the existence of a bug, trace its cause, or verify a fix. In order to alleviate this, PlanWorks has the ability to execute the planner on-line, breakpoint, and write only specified steps.

This planner control mechanism is achieved through the EUROPA$_2$ notion of a model as a compiled shared library. From within PlanWorks, the model, planner, and initial state are initialized and the user is presented with a control panel offering the ability to execute the next step and write, execute and write the next n steps, execute the next n steps without writing, execute to the end and write the final plan, or terminate the current run. Execution causes dynamic updates of the Sequence Steps View, ensuring that the user has an up-to-date view of what the planner is doing.

Beyond this, because models in EUROPA$_2$ are shared objects and initial states are files loaded at planner execution time, both can be swapped for different models or states without re-starting PlanWorks.

Aver

Introduction

"Aver" is a language for specifying run-time tests to verify proper behavior of a planning system, from the plan database to the model to the planner. It allows the description of partial or complete plans and events that occur during planning that constitute correct behavior. Files containing tests in Aver are converted to XML, which is then compiled to an internal byte-code and executed at planner run-time.

Aver is used to define tests over a *sequence of steps*, each corresponding to a partial plan logged by a planner during search. This assumption is very generic, as the planner can use any form of search from backtracking to local search. Furthermore, the planner can log plans periodically, e.g. every 5^{th} decision the planner makes.

```
Test('BasicAssertionExample',
//should be true at the beginning
  At first step : 1 == 1;
//should be true at the end as well
  At last step : 1 == 1;
//doomed to fail after the third step
  At each step > 3 : 0 != 0;
//only needs to be true once
  At any step in [0 3] :
    Count({1 2 3 4}) == 4;
//must be true at steps 3, 5, 7, and 9
  At step in {3 5 7 9} : 1 == 1;
);
```

Figure 1: Basic assertions in Aver. The first two assertions show the use of "first" and "last" in specifying steps. The third assertion specifies a subset of steps. The last two assertions show the differences between the "each" and "any" semantics.

Tests and Assertions

The largest unit of Aver is the *test*. Tests are named to allow for selective execution and contain sets of tests or *assertions*. An assertion consists of a specification of the set of steps at which the assertion must hold followed by a boolean assertion about the plan state.

A *step specification* consists of a specification of a subset of the sequence of steps, with an additional predicate of "any" or "each". An assertion with the "each" predicate must be true at all steps matching the step specification for the assertion to be considered true, assertions with the "any" predicate must be true for at least one step matching the specification. "Each" semantics is assumed if the predicate is omitted. Aver also has two special step identifiers, "first" and "last", to refer to those steps logically rather than numerically.

The boolean part of an assertion is a combination of queries for plan entites, built-in function calls, value specifications, and comparisons. All values in Aver are represented as *domains*; sets of values represented as either enumerations (i.e. "{1 2 3 4}") or intervals (i.e. "[1 4]" or "[2.5 2.9]"). Domains that contain only one value or whose upper and lower bounds are equal are called *singleton* domains. This is done because, most often, values specified in Aver are compared with the values of EUROPA$_2$ variables, which are themselves domains. Figure 1 offers some trivial example assertions.

Queries and Functions

Aver provides direct queries for three types of EUROPA$_2$ plan entities: objects, tokens, and transactions. These queries allow for the definition of subsets of entities in the partial plans matching the step specification through the specification of relevant properties of the entity type. The "Objects" query can be restricted by the object name or the values of object variables. The "Tokens" query can be restricted by the predicate name or the values of the temporal

```
Test('AnotherExample',
//there should be tokens in the plan
  At last step : Count(Tokens()) > 0;
//no backtracking in this plan
  At each step :
    Count(Transactions(type ==
      'RETRACTION')) == 0;
//a rover can't exceed the speed of
//light after the 10th step
At step > 10: Property('m_maxSpeed',
                    Objects(name ==
                        'SpiritRover'))
  < 300000000;
//there is only one location the rover
//can be at initially
  At first step :
    Count(Property('m_location',
      Tokens(predicate == 'Rover.at'
      object == 'SpiritRover'
      start == 0)))
  == 1;
);
```

Figure 2: Some more complex assertions. The first assertion uses the "Count" function and a query on the set of Transactions to ensure that no backtracking occurred during planning. The second assertion uses the "Property" function and a query on the set of Objects to ensure that a property holds. The last assertion demonstrates a query on the set of Tokens to check a property of the initial state.

or parameter variables. The "Transactions" query can be restricted by the exact name of the transaction, the type of the transaction, or the object transacted upon.

Aver has three built-in functions: "Count", "Entity", and "Property". "Count" returns the number of entities in its domain argument. "Entity" returns the nth entity in its domain argument, and "Property" returns the domain of the named variable of its single entity argument. The semantics of "Entity" are defined only for finite ordered domains, and the semantics of "Property" are only defined for single entities. Figure 2 has some more complex examples of assertions using queries and functions.

All boolean operators in Aver are defined at the level of domains, so Aver supports the usual equality, less than, greater than, less than or equal, and greater than or equal comparison operators as well as set subset-of, intersection, and exclusion operators.

A rough analogy can be drawn between Aver assertions and the `assert()` facility available in many programming languages. The common `assert()` marks a condition that must be true at a location determined by its position in code, and an Aver assertion marks a condition that must be true at a location determined by its step specification. Also, both indicate upon failure a problem that needs to be examined with a second tool; with `assert()`, this is a debugger, with Aver, PlanWorks.

Application

To demonstrate these tools, we present a model of the "pipesworld" domain, described in detail in (Milidiú, dos Santos Liproace, & de Lucena 2003), developed for EUROPA$_2$ in the modeling language developed for it, NDDL (New Domain Description Language).

Pipesworld is a domain describing the behavior of the systems used to store and transport petroleum derivative products. The peculiar constraints in this domain are:

1. The pipes must be pressurized (full) at all times.
2. The tanks have per-product capacities.
3. Because of (1), and the fluid nature of the products, it is economical to have only specific combinations of products present in a pipe simultaneously.

Products can be shifted onto a pipe from either end, forcing the product present in the pipe at the opposite end into the tank at that end.

The petroleum products are transported in units called "batches." We chose to represent a batch as a timeline with predicates representing its status in a pipe or tank or being shifted from a tank to a pipe, or vice-versa with parameters for the tank or pipe.

We chose to model only so-called "unitary" pipes—pipes that contain only one batch at a time—in the interest of simplicity. The model is, however, still interesting because there is an intermediate time between when the old batch is in the tank and the new batch occupies the pipe in which both batches are partially present in the pipe. We represent pipes as an extension of timelines, which offer automatic mutual exclusion, that are parameterized on the two tanks they connect.

Finally, tanks are represented as objects containing collections of EUROPA$_2$ resources, one for each type of product, each of which is parameterized with the number of batches of the particular product that tank can hold.

Moving a batch from a pipe to a tank creates a consumption transaction on the tank's appropriate batch-capacity resource at it's end time and moving a batch from a tank to a pipe creates a production transaction on the tank's batch-capacity resource at it's end time. The semantics of a resource in EUROPA$_2$ ensure that capacity is never exceeded.

In our initial state, there are three identical tanks; A_1, A_2, and A_3. There are two pipes, one connecting A_1 and A_2, called S_{12}, and one connecting A_1 and A_3, called S_{13}. There are also 14 batches of various products, two of which start in the pipes an the rest are in tanks.

The details of the initial and goal states are fairly uninteresting, but for the purposes of this discussion, we point out that batch 12 begins in A_3 and should end in A_2. Having constructed the model and the initial and goal states, we constructed the test in Aver before knowing what the final plan looks like.

The most trivial aspects of the Aver test confirm that the initial and goal states of the test are present in the final plan. To compose the rest of the test, we had to consider the model in conjunction with the initial and goal states. While it isn't currently possible to test the direct application of model

```
At last step :
  Count(Tokens(predicate='Batch.inPipe'
               object = 'B12'
               variable(name = 'm_pipe'
                        value= 'S13')
               start >
         Property('end',
         Tokens(predicate='Batch.inTank'
                start = 0 object = 'B12'
                variable(name = 'm_tank'
                         value = 'A3'))))))
  > 0;
```

Figure 3: A rule-checking assertion.

rules, it is possible to make assertions about their necessary effects, and it is this type of assertion that composes the majority of the test suite. For example, the assertion in Figure 3 checks the property that batch 12 must be in pipe S_{13} sometime after it's in tank A_3, which it must necessarily be to end in tank A_2.

We mention this assertion in particular because it was the first to fail. Inspection in PlanWorks confirms this. A look at the Timeline View shows that batch B12 is shifted from tank A_3 to pipe S_{12}, a clear violation of the intended semantics of the model. Images from the Constraint Network View are shown in Figure 4 to make the parameter values more visible. This indicates a missing constraint.

Looking at the model text in Figure 5, we see that there is, indeed, a missing constraint.

This constraint can be achieved using NDDL's existential quantification, which selects objects based on filtering criteria. If we add the code in Figure 6 to the rule, where the comment about the missing constraint occurs, the test should pass. And, indeed, we find that it does. This is further confirmed by PlanWorks as seen in Figure 7.

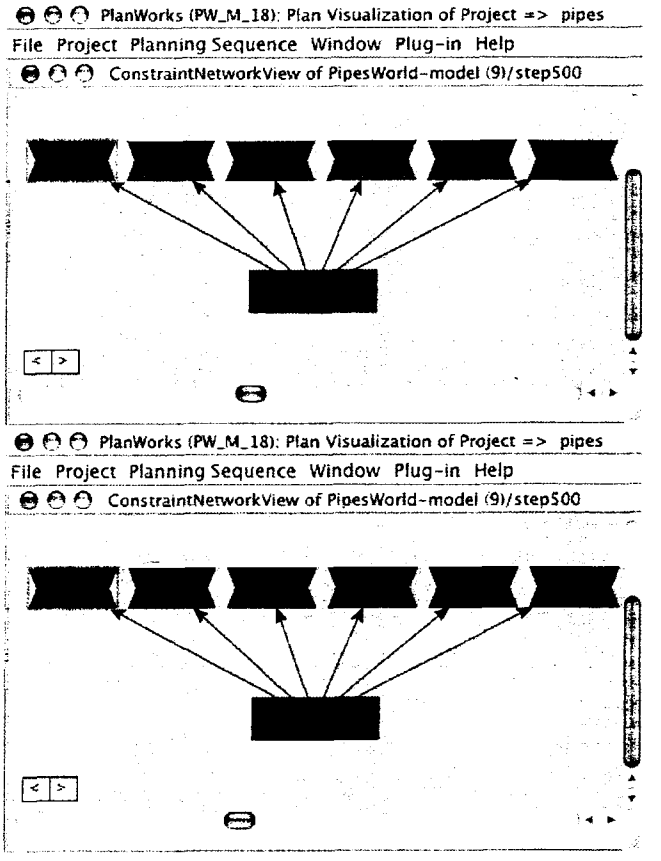

Figure 4: Top: The inTank token. Bottom: The erroneous inPipe token.

Future Work

PlanWorks

PlanWorks was originally conceived of as an integrated development environment for building and managing projects with EUROPA$_2$ and it is our intention to continue to develop features to aid in those tasks. In the near future, PlanWorks will be extended to handle model visualization and visual model building, and visualizing simple temporal networks. We also will use PlanWorks' plugin system to create planner-specific views of decision structures and heuristics.

We believe that features like automated examination of the constraint network and its execution trace to determine nogoods and the ability to alter the plan state during planner execution through the planner control mechanism will greatly add value.

Aver

As Aver becomes a more integral part of EUROPA$_2$'s test suite, we will add features to extend it's power. In particular, extending the step specification to deal with properties of

```
Batch::inTank {
  meets(object.shiftingToPipe stp);
  //should be a constraint here
  //requiring that the pipe
  //have the current tank as an endpoint
  starts(Resource.change tx)
  eq(tx.quantity, 1);
  if(object.m_product == lco) {
    eq(tx.object, m_tank.m_lco);
  }
  if(object.m_product == gasoline) {
    eq(tx.object, m_tank.m_gasoline);
  }
  //...
}
```

Figure 5: An erroneous rule.

```
bool b;
if(b == true) {
  PipeSegment p1 : {
    eq(p1.m_to, m_tank);
  }
  eq(stp.m_pipe, p1);
}
if(b == false) {
  PipeSegment p2 : {
    eq(p2.m_from, m_tank);
  }
  eq(stp.m_pipe, p2)
}
```

Figure 6: Existential quantification to fix the model.

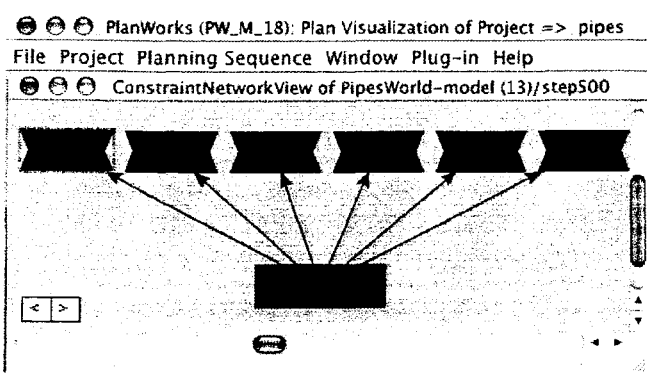

Figure 7: The inPipe token correctly constrained.

the step beyond just its number would reduce the fragility of Aver tests as well as allowing for implicative assertions, which are much more useful when verifying models.

We will extend the query capabilities to include structural assertions (entities with properties X are connected to things with properties Y), add configurable transaction sets to allow querying for custom transactions, and allow queries based on the model types of entities.

The assertion mechanism will be improved to allow for arithmetic expressions and disjunctive assertions, as well as optional assertions.

Acknowledgments

The authors would like to acknowledge Andrew Bachmann's contributions to the NDDL language used to describe EUROPA$_2$ planning domains, Will Taylor for his work on PlanWorks, Sailesh Ramakrishnan for his contributions as PlanWorks' prototype user, Bob Kanefsky for the Potato prototype that ultimately evolved into Planworks, and Mitchell Chang for his work with Conor McGann on the Tiny Test Language, which heavily informed the design of Aver.

References

Frank, J., and Jónsson, A. K. 2003. Constraint based attribute and interval planning. *Journal of Constraints*.

Milidiú, R. L.; dos Santos Liproace, F.; and de Lucena, C. J. P. 2003. Pipesworld: Planning pipeline transportation of petroleum derivatives. In *ICAPS Workshop on the Competition*.